2003

A Concurrent Logical Framework: The Propositional Fragment

Kevin Watkins[1], Iliano Cervesato[2], Frank Pfenning[1], and David Walker[3]

[1] kw,fp@cs.cmu.edu, Department of Computer Science, Carnegie Mellon University
[2] iliano@itd.nrl.navy.mil, ITT Industries, AES Division
[3] dpw@cs.princeton.edu, Department of Computer Science, Princeton University

Abstract. We present the propositional fragment CLF_0 of the Concurrent Logical Framework (CLF). CLF extends the Linear Logical Framework to allow the natural representation of concurrent computations in an object language. The underlying type theory uses monadic types to segregate values from computations. This separation leads to a tractable notion of definitional equality that identifies computations differing only in the order of execution of independent steps. From a logical point of view our type theory can be seen as a novel combination of lax logic and dual intuitionistic linear logic. An encoding of a small Petri net exemplifies the representation methodology, which can be summarized as "*concurrent computations as monadic expressions*".

1 Introduction

A logical framework is a meta-language for deductive systems. It is usually defined as a formal meta-logic or type theory together with a representation methodology. A single implementation of a logical framework can then be used to study a variety of deductive systems, thereby factoring the effort that would be required to implement each deductive system separately. Applications of logical frameworks lie mostly in logic and programming languages, where deductive systems have become a common conceptual tool and presentation device. Examples are rules of logical inference, typing rules, and rules specifying the operational semantics of a programming language. Tasks carried out with the help of logical frameworks include proof checking, proof search, and establishing meta-theoretic properties of deductive systems. For an overview and introduction to logical frameworks, their applications, and further pointers to the literature see [4, 32, 29].

The language features provided by a logical framework have a major impact on each task it supports. The right features can help make representation of deductive systems clear, direct, concise, and therefore easy to read and understand. Such elegance can, in turn, make an enormous difference when it comes to proof checking, proof search, and constructing meta-theoretic proofs. Still, each feature we add to a logical framework must be well justified as the design effort is significant and a robust framework must satisfy many subtle properties. Hence, to design an effective framework, we should identify features that most effectively support recurring idioms in the definition and manipulation of deductive systems.

Some of the most commonly recurring concepts in deductive systems are *parameterization* and *variable binding*: quantified formulas are pervasive in logic; programming languages contain parameterized expressions such as functions, objects, modules, and others; and inference rules and deductions themselves are often parameterized. LF [18] and other frameworks provide intrinsic support for parameterized objects through dependent functions. Common tasks such as renaming variables and substitution need not be coded up explicitly, as they are handled automatically by the framework when the appropriate representation strategy is chosen. With this support, simple phenomena such as α-convertibility and syntactic substitution have simple representations in the framework, so users of the framework can focus their efforts on truly complex phenomena of the system under investigation.

With dependent functions alone, however, representation of *stateful* programming languages can be clumsy and complex. In order to better accommodate reasoning with state, LF has been extended with selected constructs from linear logic, giving rise to the logical frameworks LLF [12] and RLF [21]. In these frameworks, users can represent state as linear hypotheses and imperative computations as linear functions, yielding more concise representations than are possible in LF. Since the state concept pervades deductive systems of many different kinds, we judge this extension to be justified, though at present there is much less practical experience with such linear frameworks.

Unfortunately, LF, as well as LLF and RLF, lack effective support for representing or manipulating systems involving *concurrency*, which has come to be nearly as pervasive as state. The obvious encodings of concurrent programming languages in LLF force a transformation of the operational semantics into continuation-passing style (see the example in Section 2.1), thereby fixing the order of all steps in a concurrent computation. This amounts to an interleaving semantics for concurrency rather than a truly concurrent one. While it is possible to develop, within the framework, explicit judgments specifying which computations should be considered equivalent, reasoning with or about such a specification can be exceedingly cumbersome.

Concurrent LF (CLF), the topic of this paper, is a new logical framework that extends LLF with additional linear constructs ($A_1 \otimes A_2$, 1, and $\exists x{:}A_1.\,A_2$) that make it possible to represent concurrent computations in a natural and convenient fashion. However, if they were added freely, these new connectives would interfere with standard representation techniques and would destroy one of the most fundamental properties of an LF-style framework: namely, that the structure of a canonical form is essentially determined by its type. To avoid these problems, we take the further step of *encapsulating* these new primitives by means of a *monad* that protects the conventional LF and LLF fragments of the framework. Within the monad, the natural equational theory of our additional operators gives rise to a notion of definitional equality that makes representations of concurrency adequate by ensuring that different interleavings of independent concurrent steps are indistinguishable. Although monads have been used to separate pure and effectful computations in functional programming languages, to the authors' knowledge this is their first use in a logical framework or theorem-proving environment to separate one logic from another.

Developing a logical framework goes beyond assembling a toolkit of useful representation mechanisms. The bulk of the effort consists of proving that the resulting language is well behaved for the purposes of both representation and computation. For example, it is highly desirable that type checking be decidable in a logical framework based on type theory. As the language expands, going from LF to LLF to CLF, the difficulty of this meta-theoretic investigation grows at an alarming rate, even for experienced researchers. In order to offset this increasing complexity the present paper also introduces a new methodology for developing the meta-theory of LF-style logical frameworks. It is based on the observation that, since LF-style representations rely exclusively on canonical forms, there is no need for the framework to define—or the meta-theory to investigate—anything but canonical forms. This is accomplished using an inductive notion of ***instantiation***, replacing normalization with respect to β-reduction used in traditional presentations.

The present paper concentrates on CLF_0, the propositional sublanguage of CLF, which already exhibits the principal phenomena concerning concurrency. The use of the framework is illustrated by an encoding of Petri-net computations, a simple but fundamental model of concurrency. The interested reader is referred to the accompanying technical reports [36, 13] for the definition of full CLF, the development of its meta-theory [36], and a number of larger examples [13]. These examples include an encoding of a version of ML that supports suspensions with memoization, mutable references, futures in the style of Multilisp [17], concurrency in the style of CML [35], and more. They also include a language for the representation of security protocols based on multiset rewriting [11], and representations of the synchronous and asynchronous π-calculus [26].

The remainder of this paper is organized as follows. In Section 2 we define CLF_0, including its syntax, typing rules, and definitional equality. Section 3 develops the meta-theory of CLF_0, proving decidability of typing and definitional equality. This is followed by a discussion of related work in Section 4 and a conclusion (Section 5) with some comments on future work.

2 Propositional CLF

We introduce the propositional fragment of the concurrent logical framework in stages. In the first stage, we briefly review the linear logical framework (LLF), its properties, and its shortcomings with respect to concurrency. The following stages describe the extensions yielding CLF, which aim to address these shortcomings.

2.1 The Linear Fragment

The propositional fragment LLF_0 of the linear logical framework [12] is based on unrestricted and linear hypothetical judgments $\Gamma; \Delta \vdash_\Sigma M : A$ where Γ is a context of unrestricted hypotheses $u{:}A$ (subject to exchange, weakening, and contraction), Δ is a context of linear hypotheses $x \hat{:} A$ (subject only to exchange), M

is an object and A is a type. The signature Σ declares the base types and constants from which objects are constructed. Under the Curry-Howard isomorphism, M can also be read as a proof term, and A as a proposition of intuitionistic linear logic in its formulation as DILL [3].

Since the signature is fixed for a given typing derivation, we henceforth suppress it for the sake of brevity. In addition, syntactic objects are considered only up to α-equivalence of their bound variables. Exchange is not noted explicitly in the typing rules, and only instances of the typing rules for which all variables in the contexts have unique names are allowed.

The LF representation methodology establishes a bijection between *canonical objects* of appropriate type and the terms and deductions of an object language to be represented. The appropriate notion of "canonical" turns out to be long $\beta\eta$-normal form. In order to define these inductively, the single typing judgment $\Gamma; \Delta \vdash M : A$ is refined into two judgments:

$\Gamma; \Delta \vdash N \Leftarrow A$	N is canonical of type A
$\Gamma; \Delta \vdash R \Rightarrow A$	R is atomic of type A

A canonical object N is an introduction form or is an atomic object of base type. An atomic object R is a sequence of elimination forms applied to a variable or constant. Further judgments check that types, contexts, and signatures are well-formed; they are omitted, being entirely straightforward for the propositional fragment.

The types of LLF_0 are freely generated from the constructors $\multimap$, $\rightarrow$, $\&$ and $\top$ and base types. These comprise the largest fragment of intuitionistic linear logic with traditional connectives for which unique canonical forms exist. This property is essential for the use of LLF_0 as a logical framework, because of the central role of canonical forms in its representation methodology. The syntax and the typing rules for the canonical variant of LLF_0 are shown in Figure 1.

Example. The Petri net in Figure 2 will serve as a running example of the various encoding techniques used in this paper. The representation of Petri nets in linear logic goes back to Martí-Oliet and Meseguer [24] and has been treated several times in the literature. Familiarity with Petri nets is assumed, and their encoding is only given by example. We shall however stress that we are adopting the "individual token philosophy" [8] by which the tokens within a place are not interchangeable. A planned extension of CLF with the notion of proof irrelevance [31, 34] would allow a direct encoding of the more mainstream "collective token philosophy". Further details may be found in the companion technical report [13].

Each place in a Petri net is represented by a type constant p. The state of the net is encoded as a collection of linear hypotheses: there is an assumption $x \hat{:} p$ for every token in place p. There is also a separate type constant X representing an (unspecific) goal state.

For each transition t there is an object constant[4]

$$t : (q_1 \multimap \ldots \multimap q_n \multimap \mathsf{X}) \multimap (p_1 \multimap \ldots \multimap p_m \multimap \mathsf{X})$$

[4] We adopt the convention that the connective $\multimap$ is right associative.

$$
\begin{array}{rl}
A, B, C ::= a \mid A \multimap B \mid A \rightarrow B \mid A \mathbin{\&} B \mid \top & N ::= \hat{\lambda} x.\, N \mid \lambda u.\, N \mid \langle N_1, N_2 \rangle \mid \langle\rangle \mid R \\
\Gamma ::= \Gamma, u{:}A \mid \cdot & R ::= c \mid u \mid x \mid R\,{}^\wedge N \mid R\; N \mid \pi_1 R \mid \pi_2 R \\
\Delta ::= \Delta, x \overset{\wedge}{:} A \mid \cdot & \Sigma ::= \Sigma, a{:}\mathsf{type} \mid \Sigma, c{:}A \mid \cdot
\end{array}
$$

$$
\frac{\Gamma; \Delta, x \overset{\wedge}{:} A \vdash N \Leftarrow B}{\Gamma; \Delta \vdash \hat{\lambda} x.\, N \Leftarrow A \multimap B} \multimap\mathbf{I}
\qquad
\frac{\Gamma, u{:}A; \Delta \vdash N \Leftarrow B}{\Gamma; \Delta \vdash \lambda u.\, N \Leftarrow A \rightarrow B} \rightarrow\mathbf{I}
$$

$$
\frac{\Gamma; \Delta \vdash N_1 \Leftarrow A \quad \Gamma; \Delta \vdash N_2 \Leftarrow B}{\Gamma; \Delta \vdash \langle N_1, N_2 \rangle \Leftarrow A \mathbin{\&} B} \mathbin{\&}\mathbf{I}
\qquad
\frac{}{\Gamma; \Delta \vdash \langle\rangle \Leftarrow \top} \top\mathbf{I}
\qquad
\frac{\Gamma; \Delta \vdash R \Rightarrow a}{\Gamma; \Delta \vdash R \Leftarrow a} \Rightarrow\Leftarrow
$$

$$
\frac{}{\Gamma; \cdot \vdash c \Rightarrow \Sigma(c)} c
\qquad
\frac{}{\Gamma; \cdot \vdash u \Rightarrow \Gamma(u)} u
\qquad
\frac{}{\Gamma; x \overset{\wedge}{:} A \vdash x \Rightarrow A} x
$$

$$
\frac{\Gamma; \Delta_1 \vdash R \Rightarrow A \multimap B \quad \Gamma; \Delta_2 \vdash N \Leftarrow A}{\Gamma; \Delta_1, \Delta_2 \vdash R\,{}^\wedge N \Rightarrow B} \multimap\mathbf{E}
\qquad
\frac{\Gamma; \Delta \vdash R \Rightarrow A \rightarrow B \quad \Gamma; \cdot \vdash N \Leftarrow A}{\Gamma; \Delta \vdash R\; N \Rightarrow B} \rightarrow\mathbf{E}
$$

$$
\frac{\Gamma; \Delta \vdash R \Rightarrow A \mathbin{\&} B}{\Gamma; \Delta \vdash \pi_1 R \Rightarrow A} \mathbin{\&}\mathbf{E}_1
\qquad
\frac{\Gamma; \Delta \vdash R \Rightarrow A \mathbin{\&} B}{\Gamma; \Delta \vdash \pi_2 R \Rightarrow B} \mathbin{\&}\mathbf{E}_2
$$

Fig. 1. The LLF$_0$ Language

expressing that the goal state X can be reached from a state with tokens in places $p_1, \ldots, p_m$ if the goal can be reached from the state with tokens in places $q_1, \ldots, q_n$ instead. Such a rule can be read as removing tokens from $p_1, \ldots, p_m$ and placing them on $q_1, \ldots, q_n$.

The initial state of the net in Figure 2 is represented by

$$\Delta_0 \;=\; r_1 \overset{\wedge}{:} \mathsf{r},\; n_1 \overset{\wedge}{:} \mathsf{n},\; n_2 \overset{\wedge}{:} \mathsf{n},\; b_1 \overset{\wedge}{:} \mathsf{b},\; b_2 \overset{\wedge}{:} \mathsf{b},\; b_3 \overset{\wedge}{:} \mathsf{b},\; a_1 \overset{\wedge}{:} \mathsf{a}$$

and the transitions are represented by the following signature.

$$
\begin{array}{ll}
\mathsf{P} : (\mathsf{r} \multimap \mathsf{X}) \multimap (\mathsf{p} \multimap \mathsf{X}) & \mathsf{A} : (\mathsf{c} \multimap \mathsf{X}) \multimap (\mathsf{b} \multimap \mathsf{b} \multimap \mathsf{a} \multimap \mathsf{X}) \\
\mathsf{R} : (\mathsf{p} \multimap \mathsf{n} \multimap \mathsf{b} \multimap \mathsf{X}) \multimap (\mathsf{r} \multimap \mathsf{X}) & \mathsf{C} : (\mathsf{a} \multimap \mathsf{X}) \multimap (\mathsf{c} \multimap \mathsf{X})
\end{array}
$$

The adequacy theorem for this representation states that:

> *Final state $q_1, \ldots, q_n$ can be reached from initial state $p_1, \ldots, p_m$ iff there is a canonical object N such that*
>
> $$\cdot; \cdot \vdash N \Leftarrow (q_1 \multimap \ldots \multimap q_n \multimap \mathsf{X}) \multimap (p_1 \multimap \ldots \multimap p_m \multimap \mathsf{X})$$
>
> *Moreover, there is a bijection between* sequences of firings *of the transition rules of the Petri net (according to the individual token philosophy) and such canonical objects.*

By forcibly distinguishing sequences of firings, the LLF$_0$ representation fails to capture the inherent concurrency of a Petri net. For example, in the state in

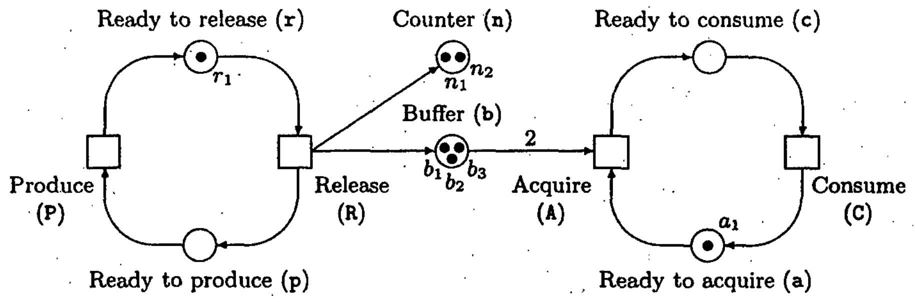

Fig. 2. A Producer/Consumer Petri Net with Labeled Marking

Figure 2, the R and A transitions can both fire, and do not interfere with each other. However, our current representation yields different terms for the two interleavings:

$$\begin{aligned}
&\cdot; \Delta_0, f \hat{:} (\mathsf{c} \multimap \mathsf{b} \multimap \mathsf{b} \multimap \mathsf{n} \multimap \mathsf{n} \multimap \mathsf{n} \multimap \mathsf{p} \multimap \mathsf{X})\\
&\quad \vdash \mathsf{R}\,\hat{}\,(\hat{\lambda} p_1.\, \hat{\lambda} n_3.\, \hat{\lambda} b_4.\, \mathsf{A}\,\hat{}\,(\hat{\lambda} c_1.\, f\,\hat{}\,c_1\,\hat{}\,b_3\,\hat{}\,b_4\,\hat{}\,n_1\,\hat{}\,n_2\,\hat{}\,n_3\,\hat{}\,p_1)\,\hat{}\,b_1\,\hat{}\,b_2\,\hat{}\,a_1)\,\hat{}\,r_1 \Leftarrow \mathsf{X}\\
&\cdot; \Delta_0, f \hat{:} (\mathsf{c} \multimap \mathsf{b} \multimap \mathsf{b} \multimap \mathsf{n} \multimap \mathsf{n} \multimap \mathsf{n} \multimap \mathsf{p} \multimap \mathsf{X})\\
&\quad \vdash \mathsf{A}\,\hat{}\,(\hat{\lambda} c_1.\, \mathsf{R}\,\hat{}\,(\hat{\lambda} p_1.\, \hat{\lambda} n_3.\, \hat{\lambda} b_4.\, f\,\hat{}\,c_1\,\hat{}\,b_3\,\hat{}\,b_4\,\hat{}\,n_1\,\hat{}\,n_2\,\hat{}\,n_3\,\hat{}\,p_1)\,\hat{}\,r_1)\,\hat{}\,b_1\,\hat{}\,b_2\,\hat{}\,a_1 \Leftarrow \mathsf{X}
\end{aligned}$$

The only way to identify these executions in LLF is to write higher-level judgments explicitly relating the representations of admissible interleavings of the same trace. This is undesirable for two reasons: first these declarations are complicated even for simple nets; second, we would need to rewrite them from scratch for every new net we consider. Note that this also forces us to abandon the propositional language LLF_0 for the dependently typed LLF.

Given how pervasive this problem is when analyzing concurrent systems, we devised an extension of LLF_0 that views executions such as the above as partial orders, identifying all of their admissible interleavings. We will describe this language, CLF_0, in the next two sections: we first introduce sufficient infrastructure to provide an alternative to the continuation-passing style of representation forced by LLF (as witnessed by the spurious goal state X). We then adjust the notion of definitional equality so that independent steps can commute.

2.2 The Monadic Fragment

A simple attempt to represent Petri nets without the continuation-passing transformation would introduce the linear logic connective $\otimes$ and its unit 1 to the framework [9]. The LLF_0 transition

$$t : (q_1 \multimap \ldots \multimap q_n \multimap \mathsf{X}) \multimap (p_1 \multimap \ldots \multimap p_m \multimap \mathsf{X})$$

would then be replaced with the more straightforward

$$t' : p_1 \otimes \ldots \otimes p_m \multimap q_1 \otimes \ldots \otimes q_n$$

However, this language does not meet the criteria we require of a *logical framework*. Modeling reachability is not enough: we also want to establish a bijection between Petri net computations and appropriately typed objects in the framework. If LLF$_0$ is extended with all (or even some) additional connectives of dual intuitionistic linear logic a number of problems establishing adequate encodings arise. The most immediate is that adding an object with any of these types can destroy the adequacy of completely unrelated encodings in the framework.

Observing the declarations

$$\begin{array}{ll} \mathsf{c} : 1 & \mathsf{z} : \mathsf{nat} \\ \mathsf{nat} : \mathsf{type} & \mathsf{s} : \mathsf{nat} \to \mathsf{nat} \end{array}$$

we see that nat contains not only terms such as z and s z but also (let $1 = \mathsf{c}$ in z). There is no longer a bijective correspondence of the type nat with the set of natural numbers.[5] Similar examples would arise in the presence of a constant of type $A \otimes B$ or $!A$. While such a language might technically be conservative over LLF$_0$, it would be impossible to embed an LLF$_0$ encoding in a larger signature using the new types—the adequacy of the LLF$_0$ encoding would be destroyed.

The underlying issue here is difficult to characterize formally, but it can be stated informally as follows: the structure of canonical forms should be *type-directed*. This leads to the inversion principles necessary to prove the adequacy of encodings. For example, we would like to know that every term of type nat is of the form z or s t where t : nat. It is easy to see that the unrestricted use of elimination forms such as (let $1 = t$ in t') subverts this principle, because the subterm t is not constrained by the type of the overall term.

In order to obtain a tractable, yet sufficiently expressive type theory we employ a technique familiar from functional programming, which does not appear to have been used in logical frameworks or theorem provers: use a monad [27] to encapsulate the effects of concurrency. This encapsulation protects the equational theory of LLF$_0$. Moreover, the notion of canonical form outside the monad extends the prior notions *conservatively*. This property of the method should not be underestimated, because it means that all encodings already devised for LF or LLF remain adequate, and their adequacy proofs can remain exactly the same!

We write $\{A\}$ for the monad type, which in lax logic would be written $\bigcirc A$ [33]. But which types should be available inside the monad? They must be expressive enough to represent the state after a computation step in the concurrent object language. This is most naturally represented by the multiplicative conjunction $\otimes$. Then our transition rule can be written

$$t'' : p_1 \multimap \ldots \multimap p_m \multimap \{q_1 \otimes \ldots \otimes q_n\}$$

where currying eliminates the use of $\otimes$ on the left-hand side. In order to cover the case $n = 0$ the multiplicative unit 1 is included. Though it does not arise in this example, a transition could also generate an element of persistent (unrestricted) type, so we also allow types $!A$. We call the new types *synchronous*, borrowing

[5] Examples such as ($\hat{\lambda} x.$ let $1 = x$ in z : $1 \multimap$ nat) show that the term above cannot simply be equal to z.

$$\begin{array}{rlcrl} & \ldots & & N ::= & \ldots \mid \{E\} \\ A,B,C ::= & \ldots \mid \{S\} & & E ::= & \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M \\ S ::= & S_1 \otimes S_2 \mid 1 \mid\ !A \mid A & & p ::= & p_1 \otimes p_2 \mid 1 \mid\ !u \mid x \\ \Psi ::= & p \mathrel{\hat{:}} S, \Psi \mid \cdot & & M ::= & M_1 \otimes M_2 \mid 1 \mid\ !N \mid N \end{array}$$

$$\ldots \quad \frac{\Gamma;\Delta \vdash E \leftarrow S}{\Gamma;\Delta \vdash \{E\} \Leftarrow \{S\}}\ \{\}\mathrm{I} \qquad \frac{\Gamma;\Delta_1 \vdash R \Rightarrow \{S_0\} \quad \Gamma;\Delta_2; p \mathrel{\hat{:}} S_0 \vdash E \leftarrow S}{\Gamma;\Delta_1,\Delta_2 \vdash (\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E) \leftarrow S}\ \{\}\mathrm{E}$$

$$\frac{\Gamma;\Delta \vdash M \Leftarrow S}{\Gamma;\Delta \vdash M \leftarrow S} \Leftarrow\leftarrow \qquad \frac{\Gamma;\Delta \vdash E \leftarrow S}{\Gamma;\Delta;\cdot \vdash E \leftarrow S} \leftarrow\leftarrow$$

$$\frac{\Gamma;\Delta; p_1 \mathrel{\hat{:}} S_1, p_2 \mathrel{\hat{:}} S_2, \Psi \vdash E \leftarrow S}{\Gamma;\Delta; p_1 \otimes p_2 \mathrel{\hat{:}} S_1 \otimes S_2, \Psi \vdash E \leftarrow S}\ \otimes\mathrm{L} \qquad \frac{\Gamma;\Delta;\Psi \vdash E \leftarrow S}{\Gamma;\Delta; 1 \mathrel{\hat{:}} 1, \Psi \vdash E \leftarrow S}\ 1\mathrm{L}$$

$$\frac{\Gamma, u{:}A;\Delta;\Psi \vdash E \leftarrow S}{\Gamma;\Delta; !u \mathrel{\hat{:}} !A, \Psi \vdash E \leftarrow S}\ !\mathrm{L} \qquad \frac{\Gamma;\Delta, x \mathrel{\hat{:}} A;\Psi \vdash E \leftarrow S}{\Gamma;\Delta; x \mathrel{\hat{:}} A, \Psi \vdash E \leftarrow S}\ \mathrm{AL}$$

$$\frac{\Gamma;\Delta_1 \vdash M_1 \Leftarrow S_1 \quad \Gamma;\Delta_2 \vdash M_2 \Leftarrow S_2}{\Gamma;\Delta_1,\Delta_2 \vdash M_1 \otimes M_2 \Leftarrow S_1 \otimes S_2}\ \otimes\mathrm{I} \qquad \frac{}{\Gamma;\cdot \vdash 1 \Leftarrow 1}\ 1\mathrm{I} \qquad \frac{\Gamma;\cdot \vdash N \Leftarrow A}{\Gamma;\cdot \vdash !N \Leftarrow !A}\ !\mathrm{I}$$

Fig. 3. The CLF$_0$ Extensions to LLF$_0$

terminology from Andreoli [2], and denote them by S. The resulting extension to the language of types is shown in Figure 3.

The language of objects is extended accordingly. The synchronous types S type *monadic expressions* E. The introduction forms M are constructors for multiplicative pairs, the multiplicative unit, and the unrestricted modality (!). The elimination form is a let binding eliminating the monad and matching the synchronous constructors against a pattern p. To our knowledge, this canonical formulation of the proof term assignment for lax logic is novel. Patterns are classified by synchronous types S and are collected into a context Ψ.

There are three typing judgments in addition to the judgments already noted for LLF$_0$:

$$\Gamma;\Delta \vdash_\Sigma E \leftarrow S \qquad\qquad \Gamma;\Delta;\Psi \vdash_\Sigma E \leftarrow S \qquad\qquad \Gamma;\Delta \vdash_\Sigma M \Leftarrow S$$

The extended language CLF$_0$ inherits all the typing rules already presented for LLF$_0$. The additional typing rules are shown in Figure 3. First, there are introduction and elimination rules for {} ({}I {}E). We can see that a monadic expression is a sequence of let forms, ending in a monadic object. Immediately after each let the pattern is decomposed into assumptions of the form $x \mathrel{\hat{:}} A$ or $u{:}A$ and the body of the let is checked. This is the purpose of the judgment $\Gamma;\ \Delta;\ \Psi \vdash E \leftarrow S$, defined by the next group of rules (⊗L 1L !L AL). These correspond to left rules in a sequent calculus. Finally, there are rules to introduce the monadic objects at the end of a sequence of {}E eliminations (⊗I 1I !I).

Example revisited. The Petri net in Figure 2 is now represented almost as in dual intuitionistic linear logic [3], except that the right-hand sides of the linear implications use the monad.

$$\begin{array}{ll} \mathsf{P} : \mathsf{p} \multimap \{\mathsf{r}\} & \mathsf{A} : \mathsf{b} \multimap \mathsf{b} \multimap \mathsf{a} \multimap \{\mathsf{c}\} \\ \mathsf{R} : \mathsf{r} \multimap \{\mathsf{p} \otimes \mathsf{n} \otimes \mathsf{b}\} & \mathsf{C} : \mathsf{c} \multimap \{\mathsf{a}\} \end{array}$$

The monadic encapsulation and the canonical forms of monadic expressions tightly constrain the form of objects constructed from this signature. Adopting α-equivalence—for the moment—as the framework's definitional equality, there is an analog of the earlier adequacy theorem.

The example firings are rewritten as follows.

$$\begin{array}{l} \cdot; \Delta_0 \vdash \{\mathsf{let}\ \{p_1 \otimes n_3 \otimes b_4\} = \mathsf{R}\hat{\ } r_1\ \mathsf{in} \\ \qquad\quad \mathsf{let}\ \{c_1\} = \mathsf{A}\hat{\ } b_1 \hat{\ } b_2 \hat{\ } a_1\ \mathsf{in} \\ \qquad\qquad c_1 \otimes b_3 \otimes b_4 \otimes n_1 \otimes n_2 \otimes n_3 \otimes p_1\} \Leftarrow \{\mathsf{c} \otimes \mathsf{b} \otimes \mathsf{b} \otimes \mathsf{n} \otimes \mathsf{n} \otimes \mathsf{n} \otimes \mathsf{p}\} \\ \cdot; \Delta_0 \vdash \{\mathsf{let}\ \{c_1\} = \mathsf{A}\hat{\ } b_1 \hat{\ } b_2 \hat{\ } a_1\ \mathsf{in} \\ \qquad\quad \mathsf{let}\ \{p_1 \otimes n_3 \otimes b_4\} = \mathsf{R}\hat{\ } r_1\ \mathsf{in} \\ \qquad\qquad c_1 \otimes b_3 \otimes b_4 \otimes n_1 \otimes n_2 \otimes n_3 \otimes p_1\} \Leftarrow \{\mathsf{c} \otimes \mathsf{b} \otimes \mathsf{b} \otimes \mathsf{n} \otimes \mathsf{n} \otimes \mathsf{n} \otimes \mathsf{p}\} \end{array}$$

With the introduction of synchronous connectives, and their encapsulation within the monadic construction, we have achieved a simple encoding of Petri nets and provided a syntax for executions that is separate from the traditional LLF_0 terms. However, α-equivalence still distinguishes the two executions above despite the fact that their R and A transitions are independent. Since the two **lets** bind and use different variables, we *should* be able to identify their permutations, with the sandboxing effect of the monad protecting the surrounding LLF_0 terms. We will now formalize this intuition.

2.3 Concurrent Equality

In essence, our objective is to identify all the usual commuting conversions between synchronous operators, but have them stop at the monadic membrane. In keeping with the philosophy espoused here of presenting the core concepts of the framework computationally, we give a direct definition of this *concurrent equality* as a decision procedure. Figure 4 shows the new syntax and inference rules associated with the definition.

The definition relies on the subsidiary concept of a *concurrent context*. As usual, the notation $\epsilon[E]$ stands for the expression constructed by replacing the hole $[]$ in ϵ with E.

The judgment $E_1 =_c E_2$ holds when E_1 and E_2 represent the same underlying concurrent computation even though their syntactic representations may differ. The rule marked (*) is subject to the side condition that no variable bound by p be free in the conclusion or bound by the context ϵ, and that no variable free in R_2 be bound by the context ϵ. Intuitively, this rule expresses that we have to find a subcomputation R_2 of the right-hand side that starts with the same step R_1 as the left-hand side. Furthermore, the remaining computation E_1 on the left-hand

$$\epsilon ::= [] \mid \mathsf{let}\ \{p\} = R\ \mathsf{in}\ \epsilon$$

$$\frac{M_1 = M_2}{M_1 =_c M_2} \qquad \frac{R_1 = R_2 \quad E_1 =_c \epsilon[E_2]}{(\mathsf{let}\ \{p\} = R_1\ \mathsf{in}\ E_1) =_c \epsilon[\mathsf{let}\ \{p\} = R_2\ \mathsf{in}\ E_2]} * \qquad \frac{E_1 =_c E_2}{E_1 = E_2}$$

(*) No variable bound by p is free in the conclusion or bound by the context ϵ, and no variable free in R_2 is bound by the context ϵ.

Fig. 4. Concurrent Equality

side must equal the remaining computation on the right-hand side, which consists of the steps preceding R_2 (in ϵ) and those following R_2 (in E_2) composed in $\epsilon[E_2]$.

There are also unmarked equality judgments $N_1 = N_2$, $R_1 = R_2$, and $M_1 = M_2$ and congruences for them (not shown). An equality judgment is not taken to mean anything in particular unless the subjects of the judgment are well typed. A typed equality judgment $\Gamma; \Delta \vdash N_1 = N_2 \Leftarrow A$ can then be defined by $(\Gamma; \Delta \vdash N_1 \Leftarrow A) \wedge (\Gamma; \Delta \vdash N_2 \Leftarrow A) \wedge (N_1 = N_2)$.

Returning to the Petri-net example developed in Section 2.2, it is easy to show that the two CLF$_0$ objects corresponding to the two different interleavings of the example Petri net execution are concurrently equal. This is crystallized as a better adequacy theorem:

Final state $q_1, \ldots, q_n$ can be reached from initial state $p_1, \ldots, p_m$ iff there is a canonical object N such that

$$\cdot; \cdot \vdash N \Leftarrow p_1 \multimap \ldots \multimap p_m \multimap \{q_1 \otimes \ldots \otimes q_n\}$$

Moreover, there is a bijection between concurrent executions *(*traces*) of the transition rules of the Petri net (according to the individual token philosophy) and equivalence classes of such canonical objects modulo $=$.*

3 Meta-theory

This section sketches the meta-theory of the canonical formulation of CLF$_0$. Additional details and a development of the dependent case may be found in the companion theory technical report [36].

3.1 Identity and substitution properties

As discussed in Section 2, the CLF$_0$ framework—and full CLF as well—syntactically restrict the form of objects so that they will always be canonical. This is a good design choice in the logical frameworks context, but it carries with it the obligation to ensure that the underlying logic (via the Curry-Howard isomorphism, if you like) is sensible. In particular, the principles of *identity* and *substitution* must hold.

Identity. *Unrestricted case*: For any Γ and A, $\Gamma, u{:}A; \cdot \vdash N \Leftarrow A$ for some N. *Linear case*: For any Γ and A, $\Gamma; x\hat{:}A \vdash N \Leftarrow A$ for some N.

Substitution. *Unrestricted case*: if $\Gamma; \cdot \vdash N_0 \Leftarrow A$ and $\Gamma, u{:}A; \Delta \vdash N \Leftarrow C$ then $\Gamma; \Delta \vdash N' \Leftarrow C$ for some N'. *Linear case*: if $\Gamma; \Delta_1 \vdash N_0 \Leftarrow A$ and $\Gamma; \Delta_2, x\hat{:}A \vdash N \Leftarrow C$ then $\Gamma; \Delta_1, \Delta_2 \vdash N' \Leftarrow C$ for some N'.

In the standard reduction-oriented treatment of proofs, these are fairly trivial, because variables and general terms are in the same syntactic category. Substitution simply syntactically replaces the target variable with the substituend—possibly creating redices. Here, redices are not syntactially allowed, and variables are syntactically *atomic* while general terms are syntactically *normal*, so it is not possible to directly replace a variable with a substituend. By the same token, a variable of higher type cannot stand by itself as a canonical object—canonical objects of higher type must be introduction forms—so the identity principle cannot be witnessed by a bare variable.

Instead, the meta-theory of CLF relies on *algorithms* that *compute* witnesses to the identity and substitution principles. These are, respectively, the *expansion algorithm* and the *instantiation algorithm*.[6]

Principle	**Algorithm**	**Supersedes**	**Notation**
Substitution	Instantiation	β-normalization	$\mathsf{inst\_n}_A(x.\, N, N_0) \equiv N'$
Identity	Expansion	η-normalization	$\mathsf{expand}_A(R) \equiv N$

Think of the instantiation operator $\mathsf{inst\_n}_A(x.\, N, N_0)$ as computing the canonical form of the result of instantiating the variable x in the object N with the object N_0. The instantiation operator is indexed by the type A of the substituend N_0. If A is a base type, we have $\mathsf{inst\_n}_A(x.\, N, N_0) = [N_0/x]N$; that is, instantiation reduces to ordinary syntactic substitution. At higher type more complex situations arise.

Dually, we think of the expansion operator $\mathsf{expand}_A(R)$ as computing the canonical form of the atomic object R of putative type A. This is analogous to η-expansion, except that the term R and its expansion inhabit different syntactic categories if A is a higher type.

These algorithms must be (and are) effectively presented, because the typing judgment of the full dependent type theory appeals to instantiation, and effective typing is central to the logical framework concept. The use of the instantiation algorithm in dependent typing has a further important ramification: the instantiation algorithm must be *effective on ill-typed terms*. Otherwise, there is a circularity between instantiation and typing, leading to a very complex meta-theory.[7] Since the substitution principle does not hold for ill-typed terms, we allow the witnessing instantiation algorithm to report failure or yield garbage on ill-typed input; e.g., $\mathsf{inst\_n}_A(x.x\ x, \lambda x.\, x\ x) \equiv \mathsf{fail}$. Garbage in, garbage out, but at least we get our garbage out in finite time!

[6] Here and in the reminder we use x generically for either a linear or unrestricted variable.
[7] This circularity, which the present treatment of CLF avoids, is analogous to the difficulties encountered in the early reduction-oriented treatments of LF, where typing refers to equality, which is decided by normalization, but normalization is only effective for well-typed terms.

$\mathsf{treduce}_A(x.\, R) \equiv B$ [Type reduction]

$$\mathsf{treduce}_A(x.\, x) \equiv A$$
$$\mathsf{treduce}_A(x.\, R\ N) \equiv C \quad \text{if } \mathsf{treduce}_A(x.\, R) \equiv B \to C$$

$\mathsf{reduce}_A(x.\, R, N_0) \equiv N'$ [Reduction]

$$\mathsf{reduce}_A(x.\, x, N_0) \equiv N_0$$
$$\mathsf{reduce}_A(x.\, R\ N, N_0) \equiv \mathsf{inst\_n}_B(y.\, N', \mathsf{inst\_n}_A(x.\, N, N_0))$$
$$\text{if } \mathsf{treduce}_A(x.\, R) \equiv B \to C \text{ and } \mathsf{reduce}_A(x.\, R, N_0) \equiv \lambda y.\, N'$$

$\mathsf{inst\_r}_A(x.\, R, N_0) \equiv R'$ [Atomic object instantiation]

$$\mathsf{inst\_r}_A(x.\, c, N_0) \equiv c$$
$$\mathsf{inst\_r}_A(x.\, y, N_0) \equiv y \quad \text{if } y \text{ is not } x$$
$$\mathsf{inst\_r}_A(x.\, R\ N, N_0) \equiv (\mathsf{inst\_r}_A(x.\, R, N_0))\ (\mathsf{inst\_n}_A(x.\, N, N_0))$$

$\mathsf{inst\_n}_A(x.\, N, N_0) \equiv N'$ [Normal object instantiation]

$$\mathsf{inst\_n}_A(x.\, \lambda y.\, N, N_0) \equiv \lambda y.\, \mathsf{inst\_n}_A(x.\, N, N_0) \quad \text{if } y \notin \mathrm{FV}(N_0)$$
$$\mathsf{inst\_n}_A(x.\, R, N_0) \equiv \mathsf{inst\_r}_A(x.\, R, N_0) \quad \text{if head}(R) \text{ is not } x$$
$$\mathsf{inst\_n}_A(x.\, R, N_0) \equiv \mathsf{reduce}_A(x.\, R, N_0) \quad \text{if } \mathsf{treduce}_A(x.\, R) \equiv a$$

Fig. 5. Instantiation, LF_0

3.2 Instantiation

Space constraints preclude the incorporation of all the cases of the definitions of these operators. Full details are available, of course, in our technical report [36].

We begin by examining the cases for the LF_0 fragment of instantiation, shown in Figure 5. The recurrence defining instantiation is based on the observation, exploited in cut elimination proofs on the logical side [30], but not so well known on the type theoretic side, that the canonical result of substituting one canonical term into another can be defined by induction on the type of the term being substituted. Accordingly, the instantiation operators are defined as a family parameterized over the type of the object being substituted. In the notation $\mathsf{inst\_c}_A(x.\, \mathsf{X}, N)$ this type A appears as a subscript. Here c is replaced by a mnemonic for the particular syntactic category to which the instantiation operator applies. The variable x is to be considered bound within the term X (of whatever category) being substituted into. The operators defined in this section should be thought of as applying to equivalence classes of concrete terms modulo α-equivalence on bound variables.

Together with the instantiation operators, and defined by mutual recursion with them, is a *reduction operator* $\mathsf{reduce}_A(x.\, R, N)$ that computes the canonical object resulting from the instantiation of x with N in the case that the *head variable* $\mathrm{head}(R)$ of the atomic object R is x. Thus, roughly speaking, it corresponds to the idea of weak head reduction for systems with β-reduction. The instantiation operator $\mathsf{inst\_r}_A(x.\, R, N)$, by contrast, is only defined if the head of R is *not* x. An-

$\mathsf{inst\_n}_A(x.\,N, N_0) \equiv N'$ [Normal object instantiation, extended]

$\mathsf{inst\_n}_A(x.\,\{E\}, N_0) \equiv \{\mathsf{inst\_e}_A(x.\,E, N_0)\}$

$\mathsf{inst\_m}_A(x.\,M, N_0) \equiv M'$ [Monadic object instantiation]

$\mathsf{inst\_m}_A(x.\,M_1 \otimes M_2, N_0) \equiv \mathsf{inst\_m}_A(x.\,M_1, N_0) \otimes \mathsf{inst\_m}_A(x.\,M_2, N_0)$
$\mathsf{inst\_m}_A(x.\,1, N_0) \equiv 1$
$\mathsf{inst\_m}_A(x.\,!N, N_0) \equiv\ !(\mathsf{inst\_n}_A(x.\,N, N_0))$
$\mathsf{inst\_m}_A(x.\,N, N_0) \equiv \mathsf{inst\_n}_A(x.\,N, N_0)$

$\mathsf{inst\_e}_A(x.\,E, N_0) \equiv E'$ [Expression instantiation]

$\mathsf{inst\_e}_A(x.\,\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E, N_0) \equiv (\mathsf{let}\ \{p\} = \mathsf{inst\_r}_A(x.\,R, N_0)\ \mathsf{in}\ \mathsf{inst\_e}_A(x.\,E, N_0))$
if $\mathrm{head}(R)$ is not x,
and $\mathrm{FV}(p) \cap \mathrm{FV}(N_0)$ is empty
$\mathsf{inst\_e}_A(x.\,\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E, N_0) \equiv \mathsf{match\_e}_S(p.\,\mathsf{inst\_e}_A(x.\,E, N_0), E')$
if $\mathsf{treduce}_A(x.\,R) \equiv \{S\}$, $\mathsf{reduce}_A(x.\,R, N_0) \equiv \{E'\}$,
and $\mathrm{FV}(p) \cap \mathrm{FV}(N_0)$ is empty
$\mathsf{inst\_e}_A(x.\,M, N_0) \equiv \mathsf{inst\_m}_A(x.\,M, N_0)$

$\mathsf{match\_m}_S(p.\,E, M_0) \equiv E'$ [Match monadic object]

$\mathsf{match\_m}_{S_1 \otimes S_2}(p_1 \otimes p_2.\,E, M_1 \otimes M_2) \equiv \mathsf{match\_m}_{S_2}(p_2.\,\mathsf{match\_m}_{S_1}(p_1.\,E, M_1), M_2)$
if $\mathrm{FV}(p_2) \cap \mathrm{FV}(M_1)$ is empty
$\mathsf{match\_m}_1(1.\,E, 1) \equiv E$
$\mathsf{match\_m}_{!A}(!x.\,E, !N) \equiv \mathsf{inst\_e}_A(x.\,E, N)$
$\mathsf{match\_m}_A(x.\,E, N) \equiv \mathsf{inst\_e}_A(x.\,E, N)$

$\mathsf{match\_e}_S(p.\,E, E_0) \equiv E'$ [Match expression]

$\mathsf{match\_e}_S(p.\,E, \mathsf{let}\ \{p_0\} = R_0\ \mathsf{in}\ E_0) \equiv \mathsf{let}\ \{p_0\} = R_0\ \mathsf{in}\ \mathsf{match\_e}_S(p.\,E, E_0)$
if $\mathrm{FV}(p_0) \cap \mathrm{FV}(E)$ and $\mathrm{FV}(p) \cap \mathrm{FV}(E_0)$ are empty
$\mathsf{match\_e}_S(p.\,E, M_0) \equiv \mathsf{match\_m}_S(p.\,E, M_0)$

Fig. 6. Instantiation, extended

$\mathsf{expand}_A(R) \equiv N$ [Expansion]

$$\mathsf{expand}_a(R) \equiv R$$
$$\mathsf{expand}_{A \multimap B}(R) \equiv \hat{\lambda} x.\,\mathsf{expand}_B(R\,\hat{}\,(\mathsf{expand}_A(x))) \quad \text{if } x \notin \mathrm{FV}(R)$$
$$\mathsf{expand}_{A \to B}(R) \equiv \lambda x.\,\mathsf{expand}_B(R\ (\mathsf{expand}_A(x))) \quad \text{if } x \notin \mathrm{FV}(R)$$
$$\mathsf{expand}_{A \& B}(R) \equiv \langle \mathsf{expand}_A(\pi_1 R), \mathsf{expand}_B(\pi_2 R)\rangle$$
$$\mathsf{expand}_{\top}(R) \equiv \langle\rangle$$
$$\mathsf{expand}_{\{S\}}(R) \equiv (\mathsf{let}\ \{p\} = R\ \mathsf{in}\ \mathsf{pexpand}_S(p))$$

$\mathsf{pexpand}_S(p) \equiv M$ [Pattern expansion]

$$\mathsf{pexpand}_{S_1 \otimes S_2}(p_1 \otimes p_2) \equiv \mathsf{pexpand}_{S_1}(p_1) \otimes \mathsf{pexpand}_{S_2}(p_2)$$
$$\mathsf{pexpand}_1(1) \equiv 1$$
$$\mathsf{pexpand}_{!A}(!x) \equiv !(\mathsf{expand}_A(x))$$
$$\mathsf{pexpand}_A(x) \equiv \mathsf{expand}_A(x)$$

Fig. 7. Expansion

other distinguishing feature is that reduction on an atomic object yields a normal object, while instantiation on an atomic object yields an atomic object.

Finally, there is a *type reduction operator* $\mathsf{treduce}_A(x.\,R)$ that computes the putative type of R given that the head of R is x and the type of x is A. Type reduction is used in side conditions that ensure that the recurrence defining instantiation is well-founded.

The recurrence defining these operators is based on a structural induction. There is an outer induction on the type subscripting the operators, and an inner simultaneous induction on the two arguments. Noting first that if $\mathsf{treduce}_A(x.\,R)$ is defined, it is a subterm of A, the fact that the recurrence relations respect this induction order can be verified almost by inspection. The only slightly subtle case is the equation for $\mathsf{reduce}_A(x.\,R\ N, N_0)$, which is the only case in which the subscripting type changes. Here the side condition $\mathsf{treduce}_A(x.\,R) \equiv B \to C$ ensures that B must be a strict subterm of A for the reduction to be defined. An instantiation such as $\mathsf{inst\_n}_A(x.\,x\ x, \lambda x.\,x\ x)$ is guaranteed to fail the side condition after only finitely many expansions of the recurrence.

Another way in which an instance of the instantiation operators might fail to be defined would be if the recursive instantiation $\mathsf{inst\_r}_A(x.\,R, N_0)$ in the same equation failed to result in a manifest lambda abstraction $\lambda y.\,N'$. In fact, this could only happen if the term N_0 failed to have the ascribed type A. So instantiation always terminates, regardless of whether its arguments are well typed, but it is not defined in all cases. After the meta-theory is further developed, it can be shown that instantiation is always defined on well-typed terms when the types match in the appropriate way.

The cases of instantiation involving the monad, shown in Figure 6, are not without interest. These lean heavily on prior work on proof term assignments for modal logics [33].

In order to extend instantiation to the full CLF_0 language, with its pattern-oriented destructor for the monadic type, it is necessary to introduce *matching operators* $\mathsf{match\_c}_S(p.\,E, \mathsf{X})$, where X is either an expression or a monadic object. The matching operator computes the result of instantiating E according to the substitution on the variables of p generated by matching p against X. (The variables in p should be considered bound in E.) In the case that X is a monadic object M_0, this is straightforward: the syntax of monadic objects corresponds precisely to that of patterns. But in the case that X is a let binding, an interesting issue arises:

$$\mathsf{match\_e}_S(p.\,\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E_1, \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E_2) \equiv ?$$

The key is found in Pfenning and Davies' non-standard substitutions for the proof terms of the modal logics of possibility and laxity [33]. These analyze the structure of the object being substituted, not, as in the usual case, the term being substituted into. The effect is similar to a commuting conversion:

$$\begin{aligned}&\mathsf{match\_e}_S(p.\,\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E_1, \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E_2) \equiv\\ &\quad(\mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ \mathsf{match\_e}_S(p.\,\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E_1, E_2))\end{aligned}$$

It is interesting that both non-standard substitution and pattern matching—the latter not present in Pfenning and Davies' system—rely in this way on an analysis of the object being substituted rather than the term being substituted into. In a sense, this commonality is what makes the harmonious interaction between CLF's modality and its synchronous types possible.

The induction order mentioned above leads immediately to the following theorem.

Theorem 1 (Definability of instantiation). *The recurrence for the reduction, instantiation, and matching operators uniquely determines the least partial functions (up to α-equivalence) solving them.*

Proof. The proof is by an outer structural induction on the type subscript, and an inner simultaneous structural induction on the two arguments. □

3.3 Expansion

The definition of expansion is shown in Figure 7. In some cases, new bound variables are introduced on the right-hand side of an equation. Any new variables in an instance of such an equation are required to be distinct from one another and from any other variables in the equation instance.

Again there is a definability theorem based on the induction order implicit in the equations.

Theorem 2 (Definability of expansion).

1. *If* $\mathsf{pexpand}_S(p_1)$ *and* $\mathsf{pexpand}_S(p_2)$ *are both defined then* p_1 *and* p_2 *are the same up to variable renaming.*
2. *Given S, there is a pattern p, fresh with respect to any given set of variables, such that* $\mathsf{pexpand}_S(p)$ *is defined.*

3. *The recurrence for expansion uniquely determines it as a total function up to α-equivalence.*

Proof. The first part is by induction on S. The second and third parts are by induction on the type subscript, using the first part to ensure that the result of $\mathsf{expand}_{\{S\}}(R)$ is unique up to α-equivalence. □

3.4 Further results

The following theorem is proved in the full generality of the dependent case in the technical report [36]. The identity and substitution principles follow immediately.

Theorem 3 (Identity and substitution principles). *The following rules are admissible.*

$$\frac{\Gamma; \Delta \vdash R \Rightarrow A}{\Gamma; \Delta \vdash \mathsf{expand}_A(R) \Leftarrow A}$$

$$\frac{\Gamma; \cdot \vdash N_0 \Leftarrow A \quad \Gamma, x{:}A; \Delta \vdash N \Leftarrow C}{\Gamma; \Delta \vdash \mathsf{inst\_n}_A(x.\,N, N_0) \Leftarrow C} \qquad \frac{\Gamma; \Delta_1 \vdash N_0 \Leftarrow A \quad \Gamma; \Delta_2, x\hat{:}A \vdash N \Leftarrow C}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{inst\_n}_A(x.\,N, N_0) \Leftarrow C}$$

Proof. By straightforward inductions. □

In the dependently-typed case, lemmas concerning the algebraic laws satisfied by expansion and instantiation (roughly analogous to confluence results) and concerning the interaction of equality and instantiation are required. Other notable theorems (which, in the dependently-typed case, are actually needed to prove the theorem above) include the following.

Theorem 4 (Decidability of equality). *Given N_1 and N_2, it is decidable whether $N_1 = N_2$.*

Proof. The formulation of the equality rules is nearly syntax-directed, so a simultaneous structural induction on the subjects of the judgment suffices. It remains only to observe that an expression can be decomposed into a concurrent context and subexpression in finitely many ways. □

Theorem 5 (Decidability of instantiation and expansion). *It is decidable whether any instance of the instantiation and expansion operators is defined, and if so, it can be effectively computed.*

Proof. For instantiation, this is proved by a simultaneous structural induction on the substituend, the term substituted into, and the putative type of the substituend. For expansion, the induction is over the structure of the type. □

Theorem 6 (Decidability of typing). *It is decidable whether any instance of the typing judgments is derivable.*

Proof. By structural induction on the subject of the judgments. □

In the dependently-typed case, the inference rules for typing are also structured in a syntax-directed manner, leading to a very simple proof of decidability [36]. This is a substantial technical improvement over prior presentations of even the LF sublanguage alone.

The interaction of equality and substitution is particularly important, since CLF's equality is where concurrency enters. Thus, the following theorems describe, in essence, how concurrent computations modeled in our framework compose.

Theorem 7. *Concurrent equality $N_1 = N_2$ is an equivalence relation.*

Proof. Reflexivity, symmetry, and transitivity can each be proved by structural inductions (with appropriate lemmas, also proved by structural induction) [36]. □

Theorem 8. *If $N = N'$ and $N_0 = N_0'$ then $\mathsf{inst\_n}_A(x.\,N, N_0) = \mathsf{inst\_n}_A(x.\,N', N_0')$, assuming one side or the other is defined.*

Proof. The proof appeals to composition laws for instantiation and a number of other technical lemmas. The inductive proofs of these lemmas and the main theorem follow the same induction order as for the decidability result [36]. □

Theorem 9. *If $R = R'$ then $\mathsf{expand}_A(R) = \mathsf{expand}_A(R')$.*

Proof. This follows by structural induction on A. □

4 Related Work

Past research has identified two main approaches to encoding concurrent computations in linear logic. Abramsky's *proofs-as-processes* [6] assumes a functional perspective where process interaction is captured by cut-elimination (normalization) steps over linear logic derivations. A second direction, which may be identified with the slogan *proofs-as-traces* (and *formulas-as-processes*), models dynamic process behaviors as proof-search, generally in the style of (linear) logic programming [24, 2, 25, 22, 14, 9].

CLF follows this second path, stressing a one-to-one correspondence between CLF proof-terms and process executions (traces) [13]. CLF differs from most of these proposals in two respects: first, it is a fully dependent logical framework, which means that it expresses not only the constructs of an object process calculus and their behavior, but also executions themselves and meta-reasoning about them. Second, the concurrent equality intrinsically supports true concurrency.

To the authors' knowledge, Honsell et. al. [20] describe the most significant application of a logical framework in the sphere of concurrency. They elegantly encode the π-calculus with substantial meta-theory in the calculus of constructions with inductive/coinductive types ($CC^{(Co)Ind}$). However, since the notion of equality of $CC^{(Co)Ind}$ does not identify permutable computations, more advanced meta-theoretic investigations would require tedious coding of an equivalence similar to CLF's concurrent equality.

The idea of monadic encapsulation goes back to Moggi's monadic meta-language [27,28] and is used heavily in functional programming. Our formulation follows the judgmental presentation of Pfenning and Davies [33], which completely avoids the need for commuting conversions, but the latter treats neither linearity nor the existence of normal forms. The exploration of monads in logic programming by Bekkers and Tarau [5] concentrates on the use of monads for data structures and all-solution predicate. This is quite different from our application and concerned neither with additional logical connectives nor a true extension of the operational semantics. Benton and Wadler [7] explore the relationship of Moggi's monadic meta-language and term calculi for linear logic with Benton's adjoint calculus, which bears some intriguing similarities with CLF. However, it is not a type theory, and the logical connectives (such as implication) common to lax logic and linear logic retain separate identities, rather than being combined, as in CLF.

The method of defining a type theory by a typed operational semantics goes back to the Automath languages [15] and has been applied to LF by Felty [16]. Our canonical formulation significantly extends and streamlines the ideas behind Felty's *canonical LF* and its extension to LLF [12]; the need for confluence and β-normalization results is eliminated. A similar philosophical outlook, but different technical realizations underly PAL+ [23] and work by Adams [1], who also consider frameworks restricted to normal forms.

5 Conclusion

In this paper, we have presented the basic design of a logical framework that internalizes parametric and hypothetical judgments, linear hypothetical judgments, and true concurrency. This supports representation of a wide variety of concepts related to logic and computation in a natural and concise manner. It also poses a host of new questions.

One of the practically important features of the linear logical framework is its operational interpretation as a logic programming language using goal-directed proof search [19,10]. We conjecture that CLF supports a conservative extension of this operational semantics. We have already constructed a representation of Mini-ML with concurrency and parallelism anticipating such an interpretation [13].

Concurrent computations in an object language are internalized as monadic expressions in CLF. The framework allows type families indexed by objects containing such expressions, which means it is possible to formulate properties of concurrent computations and relations between them. Examples are safety and possibly liveness properties, bisimulations, and other translations between models of computations.

Petri nets and other case studies have shown that, in many cases, computations should be indistinguishable also when threads interact over isomorphic objects. It appears that this can be achieved by integrating the notion of proof irrelevance [31, 34] within CLF. Once this extension has been fully worked out, CLF would be able to provide an adequate representation to Petri nets under the collective token philosophy, for example.

References

1. Robin Adams. A modular approach to logical frameworks. Talk at the TYPES 2003 Workshop, Torino, Italy, April 2003.
2. Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.
3. Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh, September 1996.
4. David Basin and Seán Matthews. Logical frameworks. In Dov Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic*. Kluwer Academic Publishers, 2nd edition, 2001.
5. Yves Bekkers and Paul Tarau. Monadic constructs for logic programming. In J. Lloyd, editor, *Proceedings of the International Logic Programming Symposium (ILPS'95)*, pages 51–65, Portland, Oregon, December 1995. MIT Press.
6. G. Bellin and P. J. Scott. On the π-calculus and linear logic. *Theoretical Computer Science*, 135:11–65, 1994.
7. P. N. Benton and Philip Wadler. Linear logic, monads, and the lambda calculus. In E. Clarke, editor, *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 420–431, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
8. Roberto Bruni and Ugo Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Information and Computation*, 156(1–2):46–89, 2000.
9. Iliano Cervesato. Petri nets and linear logic: a case study for logic programming. In M. Alpuente and M. I. Sessa, editors, *Proceedings of the 1995 Joint Conference on Declarative Programming — GULP-PRODE'95*, pages 313–318, Marina di Vietri, Italy, 1995.
10. Iliano Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, February 1996.
11. Iliano Cervesato. Typed MSR: Syntax and examples. In V.I. Gorodetski, V.A. Skormin, and L.J. Popyack, editors, *Proceedings of the First International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security — MMM'01*, pages 159–177, St. Petersburg, Russia, 21–23 May 2001. Springer-Verlag LNCS 2052.
12. Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information & Computation*, 179(1):19–75, November 2002.
13. Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002.
14. Jawahar Lal Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, May 1995.
15. N.G. de Bruijn. Algorithmic definition of lambda-typed lambda calculus. In G. Huet and G. Plotkin, editors, *Logical Environment*, pages 131–145. Cambridge University Press, 1993.
16. Amy Felty. Encoding dependent types in an intuitionistic logic. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*, pages 214–251. Cambridge University Press, 1991.
17. Robert H. Halstead. Multilisp: A language for parallel symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.
18. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

19. Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.
20. Furio Honsell, Marino Miculan, and Ivan Scagnetto. Pi-calculus in (co)inductive type theories. *Theoretical Computer Science*, 253(2):239–285, 2001.
21. Samin Ishtiaq and David Pym. A relevant analysis of natural deduction. *Journal of Logic and Computation*, 8(6):809–838, 1998.
22. Naoki Kobayashi and Akinori Yonezawa. ACL — A concurrent linear logic programming paradigm. In D. Miller, editor, *Proceedings of the 1993 International Logic Programming Symposium*, pages 279–294, Vancouver, Canada, 1993. MIT Press.
23. Zhaohui Luo. PAL+: A lambda-free logical framework. *Journal of Functional Programming*, 13(2):317–338, 2003.
24. N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic through categories: A survey. *Journal on Foundations of Computer Science*, 2(4):297–399, December 1991.
25. Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the Workshop on Extensions of Logic Programming*, pages 242–265. Springer-Verlag LNCS 660, 1992.
26. Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
27. Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings of the Fourth Symposium on Logic in Computer Science*, pages 14–23, Asilomar, California, June 1989. IEEE Computer Society Press.
28. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
29. Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.
30. Frank Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000.
31. Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 221–230, Boston, MA, June 2001. IEEE Computer Society Press.
32. Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.
33. Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.
34. Jason Reed. Proof irrelevance and strict definitions in a logical framework. Technical Report CMU-CS-02-153, Computer Science Department, Carnegie Mellon University, 2002.
35. John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
36. Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002.